Monte Carlo Algorithms for the Detection of Necessary Linear Matrix Inequality Constraints

Shafiu Jibrin

Department of Mathematics and Statistics, Northern Arizona University, Flagstaff, Arizona 86011, USA, E-mail: Shafiu.Jibrin@nau.edu

Irwin S. Pressman

School of Mathematics and Statistics, Carleton University, Ottawa, Ontario K1S 5B6, Canada, E-mail: irwin_pressman@carleton.ca

(Received February 2000, Revised September 2000)

Abstract

We reduce the size of large semidefinite programming problems by identifying necessary linear matrix inequalities *(LMI's)* using Monte Carlo techniques. We describe three algorithms for detecting necessary LMI constraints that extend algorithms used in linear programming to semidefinite programming. We demonstrate that they are beneficial and could serve as tools for a semidefinite programming preprocessor.

A *necessary* LMI is one whose removal changes the feasible region defined by all the LMI constraints. The general problem of checking whether or not a particular LMI is necessary is NP-complete. However, the methods we describe are polynomial in each iteration, and the number of iterations can be limited by stopping rules. This provides a practical method for reducing the size of some large Semidefinite Programming problems before one attempts to solve them. We demonstrate the applicability of this approach to solving instances of the Löwner ellipsoid problem. We also consider the problem of classification of all the constraints of a semidefinite programming problem as redundant or necessary.

1 Introduction

Let $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$ be $m \times m$ symmetric matrices. $\mathbf{A}$ is called *positive definite* if all its eigenvalues are strictly positive. $\mathbf{A}$ is called *positive semidefinite* if all its eigenvalues are nonnegative and at least one is zero. If $\mathbf{A}$ is positive semidefinite (respectively positive definite), we write $\mathbf{A} \succeq 0$ (respectively $\mathbf{A} \succ 0$). The symbol $\succeq$ is the *Löwner partial order* for real matrices, i.e., $\mathbf{A} \succeq \mathbf{B}$ if and only if $\mathbf{A}$-$\mathbf{B}$ is positive semidefinite.

A *semidefinite program* (SDP) is the programming problem

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ s.t. \quad & \mathrm{A}^{(j)}(\mathbf{x}) := \mathbf{A}_0^{(j)} + \sum_{i=1}^{n} x_i \mathbf{A}_i^{(j)} \succeq 0, \\ & j = 1, 2, \ldots, q \end{aligned} \tag{1.1}$$

where $\mathbf{A}_i^{(j)}$, $i = 0, 1, \ldots, n$ are $m_j \times m_j$ symmetric matrices and $\mathbf{x}, \mathbf{c} \in \mathbf{R}^n$. We assume $m_1 \leq m_2 \leq \ldots \leq m_q$. $\mathbf{R}^n$ is called the *ambient* space of the SDP problem. A single constraint (1.1) is called a *linear matrix inequality* (LMI) or *semidefinite constraint.*

1.0.1 Definitions and Four Special Assumptions

The *feasible* $\mathcal{R}$ region of the SDP problem defined by

$$\mathcal{R} = \{\mathbf{x} \mid \mathrm{A}^j(\mathbf{x}) \succeq 0, \quad 1 \leq j \leq q\}$$

is called *full-dimensional* if it has a non-empty interior. We assume (I) that $\mathcal{R}$ is bounded and full. We assume (II) that all constraints here are *strictly feasible*, i.e., for some $\mathbf{x} \in \mathbf{R}^n$, $\mathrm{A}^j(\mathbf{x}) \succ 0$. The LMI constraints (1.1) are all positive definite at a point if and only if the point is in the interior [1]. A feasible point $\mathbf{x}$ is on the boundary of the feasible region defined by the $j'th$ constraint if the determinant $\det(\mathrm{A}^j(\mathbf{x})) = 0$ or equivalently, if $\mathrm{A}^j(\mathbf{x})$ has a zero eigenvalue. For $k = 1, 2, \ldots, q$, define the regions $\mathcal{R}_k$ ($\mathcal{R} \subseteq \mathcal{R}_k$) by

$$\begin{aligned} \mathcal{R}_k \quad = \quad & \{\mathbf{x} \mid \mathrm{A}^j(\mathbf{x}) \succeq 0, \\ & j \in \{1, 2, \ldots, k-1, k+1, \ldots, q\}\}. \end{aligned}$$

Definition 1.1 $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ *is called redundant with respect to the set* $\{\mathrm{A}^j(\mathbf{x}) \succeq 0\}_{j=1}^{q}$ *if* $\mathcal{R} = \mathcal{R}_k$, *and is called necessary if* $\mathcal{R} \subset \mathcal{R}_k$.

That is, an LMI constraint is called *necessary* if its removal changes the feasible region of the problem, otherwise it is called *redundant.* A *duplicate* LMI can be both redundant and necessary depending on the order of the removal. We assume that (III) there are no *duplicate* constraints and that (IV) no pair of constraints intersect on a subset of the boundary of the feasible region of *Lebesgue measure* greater than zero. Assumption (IV) is a nontrivial requirement in general! For instance,

$$\begin{aligned} \mathrm{A}^{(1)}(\mathbf{x}) \quad := \quad & \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \\ & + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0. \end{aligned}$$

is a constraint that is the combination of two simple linear constraints:

$$A^{(2)}(\mathbf{x}) := x_1 \leq 5 \text{ and } A^{(3)}(\mathbf{x}) := x_2 \leq 2$$

so that $A^{(1)}(\mathbf{x}) \succeq 0$ and $A^{(2)}(\mathbf{x}) \succeq 0$ share a vertical boundary line below $x_2 = 2$.

1.0.2 Survey of Literature and Complexity Issues

The semidefinite programming problem (SDP) is a convex optimization problem since the feasible region $\mathcal{R}$ is convex [27]. Problems of this type arise directly in Control Theory, Statistics and Combinatorial Optimization ([1], [27]). SDP generalizes linear programming (LP) and quadratically constrained quadratic programming (QCQP) [27]. Several approaches to SDP duality generalize LP duality ([1], [23], [27], [29]). LP interior point methods can be generalized to convex programming problems with readily computable *self-concordant* barrier functions such as SDP [22] and polynomial time methods have been developed for solving SDP ([1], [26], [27]).

We make the simplifying *assumption* that $m = m_1 = m_q$. A general-purpose SDP solver like *SDPSOL*© [30] amalgamates all the constraints into a single large $m_q \times m_q$ constraint. With this representation of the problem, given a feasible near-central solution with duality gap at most $1/\epsilon$, one can find an optimal solution with duality gap less than ϵ in $O(\sqrt{m_q q}\ln(1/\epsilon))$ iterations, each requiring $O(qnm_q^3 + qn^2m_q^2)$ flops in exact arithmetic [21]. Thus, focusing on q, the number of constraints, work is $O(q^{1.5})$.

For q large, there are significant savings in computation time to be gained if the number of constraints can be effectively reduced. Memory requirements per iteration also increase with q, and can exceed a machine's capacity. As a result, SDP software such as SDPSOL and SDPpack© [2] are restricted to problems of moderate size [4]. The dual of SDP (1.1) is:

$$\begin{aligned} \max \quad & -\sum_{j=1}^{q} \mathbf{A}_0^{(j)} \bullet \mathbf{Z_j} \\ s.t. \quad & \sum_{j=1}^{q} \mathbf{A}_i^{(j)} \bullet \mathbf{Z_j} = c_i, \ i = 1, 2, \ldots, n \\ & \mathbf{Z_j} \succeq 0, \quad j = 1, 2, \ldots, q \end{aligned}$$

where the $m_j \times m_j$ symmetric matrices $\mathbf{Z_j}$ are the variables. SDPpack has a preprocessor that eliminates redundant linear constraints in the dual of the SDP by a **QR** factorization of $\mathbf{A}^T$ where $\mathbf{A}$ is the constraint matrix, i.e., $\mathbf{A}^T = \mathbf{QR}$ [20]. Diagonal elements which are zero or very small indicate which constraints are redundant and can be eliminated. Our intent is to offer tools to eliminate most redundant LMI constraints.

In the special case of *linear* constraints, methods for identifying redundant constraints include LP based deterministic methods [19] which require solving an LP to determine whether

or not a constraint is redundant. A different approach is the probabilistic *Hit-and-Run* method which identifies necessary constraints [8] by generating random lines through the interior. Each line meets the boundary at two points (*hit points*) and with probability one identifies two necessary constraints. Any constraint that is *binding* at any of the hit points is necessary. After a finite number of iterations. the identified constraints are necessary, though some of them may have been missed. Hit-and-run was first published in [8] under the name *preduce*. Three variations of the Hit-and-Run method are the hypersphere direction method (HD) [8], the coordinate direction method (CD) [19] and the Stand-and-Hit method (SH) ([17], [13]). The total time for complete identification of all necessary constraints for each of these methods is problem dependent [13].

It is difficult to extend all LP-based redundancy methods for linear constraints to LMI constraints. For example, the *Turnover Lemma* [9], applied to linear systems, replaces an inequality $\mathbf{a}^T\mathbf{x} + \mathbf{b} \geq 0$ by its complement $\mathbf{a}^T\mathbf{x} + \mathbf{b} < 0$. If the new system is infeasible, then constraint $\mathbf{a}^T\mathbf{x}+\mathbf{b} \geq 0$ is redundant; otherwise it is necessary. In the case of LMI systems, we wouldn't obtain a convex constraint from the complement of $\mathbf{A_0} + \sum_{i=1}^{n} x_i\mathbf{A_i} \succ 0$, i.e., $-(\mathbf{A_0} + \sum_{i=1}^{n} x_i\mathbf{A_i}) \succ 0$.

Here, we extend SH to the Semidefinite Stand-and-Hit (SSH) method. In some situations, we reduce computation further by the application of the Undetected-First Rule (UFR) [13]. We denote SSH combined with UFR as SSH(UFR).

1.1 The Löwner Ellipsoid test problems

The test problems introduced here are all Löwner ellipsoid problems [10] whose specifics are given in Table 1. They are of the form:

$$\begin{aligned}
&\max \log\det(\mathbf{X}) \\
&s.t. \begin{bmatrix} \mathbf{I} & \mathbf{X}\mathbf{a}^j + \mathbf{y} \\ (\mathbf{X}\mathbf{a}^j + \mathbf{y})^T & 1 \end{bmatrix} \succeq 0 \\
&(1 \leq j \leq q), \\
&\mathbf{X} \succ 0,
\end{aligned}$$

where the variable $\mathbf{X}$ denotes a $p \times p$ symmetric matrix, the variable $\mathbf{y}$ belongs to $\mathbf{R}^p$ and $\mathbf{a}^j$ are given vectors in $\mathbf{R}^p$. The entries of each vector $\mathbf{a}^j$ are generated *randomly* from the uniform distribution on the interval (0,1). We note that each of the constraints can be expressed as an LMI of the form (1.1) with $n = p(p+1)/2 + p$ variables and $m_j = p + 1$.

The test problems are used for convenience only and our results do not depend on them. We select these problems because [i] they are feasible, [ii] they have large redundancy when q is large, and [iii] we have shown elsewhere [18] that the feasible region is bounded when

Problem	p	n	m_j	q	q_{nec}
Löwner1	3	9	4	21	14
Löwner2	3	9	4	501	29
Löwner3	5	20	6	41	27
Löwner4	10	65	11	21	16
Löwner5	2	5	3	1001	11
Löwner6	2	5	3	5001	12
Löwner7	7	35	8	51	46
Löwner8	10	65	11	31	29
Löwner9	2	5	3	1001	16
Löwner10	12	90	13	36	35

Table 1: Test Problems

the vectors $\mathbf{a}^{(j)}$ are affinely independent. Our general results are *not* restricted to Löwner type problems.

The number q_{nec} of nonredundant constraints in each problem listed in Table 1 was estimated using repeated exhaustive random searches using the *SHD method* for at least 24 hours. There is a small probability that not every nonredundant constraint was found. All computations described here were performed on a PENTIUM 233 MHz using codes written in MATLAB 5.0.

1.2 The Semidefinite Redundancy Problem and SDP simplification

The smallest and the largest eigenvalues of an $m \times m$ symmetric matrix $\mathbf{A}$ are denoted by $\lambda_{\min}(\mathbf{A})$ and $\lambda_{\max}(\mathbf{A})$. The eigenvalues are denoted in descending order by $\lambda_{\max}(\mathbf{A}) = \lambda_1(\mathbf{A}) \geq \lambda_2(\mathbf{A}) \geq \ldots \geq \lambda_m(\mathbf{A}) = \lambda_{\min}(\mathbf{A})$.

The *semidefinite redundancy problem* is to decide whether or not the $k'th$ constraint $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ is redundant with respect to the set $\{\mathrm{A}^{(j)}(\mathbf{x}) \succeq 0\}_{j=1}^{q}$. The following theorem gives a necessary and sufficient condition for determining whether the $k'th$ LMI constraint is redundant. It turns a redundancy problem into an eigenvalue problem.

Theorem 1.1 $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ *is redundant if and only if the semidefinite program*

$$SDP_k: \quad \min \ \lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{x})) \quad (1.2)$$
$$s.t. \quad \mathbf{x} \in \mathcal{R}_k$$

has an optimal solution, say $\mathbf{x}^*$, *satisfying* $\lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{x}^*)) \geq 0$.

Proof: If $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ is redundant, then for every $\mathbf{x} \in \mathcal{R}_k$, $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ and $\lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{x})) \geq 0$. Hence, SDP_k has an optimal solution $\mathbf{x}^*$ satisfying $\lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{x}^*)) \geq 0$. Conversely, suppose $\mathbf{x}^*$ is an optimal solution of the SDP_k and that $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ is necessary. Then, for some $\mathbf{y} \in \mathcal{R}_k$, we have $\lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{y})) < 0$. Therefore $\lambda_{\min}(\mathrm{A}^{(k)}(\mathbf{x}^*)) < 0$, and $\mathrm{A}^{(k)}(\mathbf{x}^*) \not\succeq 0$. □

Corollary 1.1 $\mathrm{A}^{(k)}(\mathbf{x}) \succ 0$ *is necessary if the semidefinite program*

$$\min \ \mathrm{Tr}(\mathrm{A}^{(k)}(\mathbf{x})) \quad (1.3)$$
$$s.t. \quad \mathbf{x} \in \mathcal{R}_k$$

has an optimal value of the objective function which is strictly negative.

Proof: If $\mathrm{Tr}(A^{(k)}(\mathbf{x}^*)) = \lambda_1(A^{(k)}(\mathbf{x}^*)) + \lambda_2(A^{(k)}(\mathbf{x}^*)) + \ldots + \lambda_{m_k}(A^{(k)}(\mathbf{x}^*)) < 0$, then $\lambda_{m_k}(A^{(k)}(\mathbf{x}^*)) = \lambda_{\min}(A^{(k)}(\mathbf{x}^*)) < 0$. Now apply Theorem 1.1. □

The function $\mathrm{Tr}(A^{(k)}(\mathbf{x}))$ is a linear function of $\mathbf{x}$, so SDP (1.3) can be solved by SDPSOL. By Corollary 1.1, if the optimal objective value of (1.3) is negative, then the *k'th* constraint is necessary. If the objective value is nonnegative, then the *k'th* constraint may or may not be necessary. It is known that $\lambda_{\min}(A^k(\mathbf{x}))$ is a continuous, concave and nondifferentiable function of $\mathbf{x}$ ([1], [15], [16]).

The problem of checking whether or not a convex quadratic constraint (CQC) is redundant with respect to a system of CQC's is known to be NP-complete [10]. Since LMI's include convex quadratic constraints [27], SDP_k is NP-complete. Therefore, it is difficult to have an SDP-based method that uses Theorem 1.1 for redundancy identification. In the special case of linear constraints the problem SDP_k (1.2) is a linear program, a fact LP-based redundancy methods exploit.

2 Semidefinite Stand-and-Hit (SSH) Algorithm

The SSH algorithm is a Monte Carlo method for detecting necessary constraints in the system (1.1). It is based on the following idea. Fix a *standing point* $\mathbf{x}_0$ in the interior of $\mathcal{R}$. A random direction $\mathbf{s}$ and $\mathbf{x}_0$ define two line segments $\{\mathbf{x}_0 + \sigma\mathbf{s} \mid \sigma \geq 0\}$ and $\{\mathbf{x}_0 - \sigma\mathbf{s} \mid \sigma \geq 0\}$ whose intersection points with the boundary of $\mathcal{R}$, called *hit points*, identify at least one necessary LMI. We continue to look for necessary constraints until a given stopping condition is satisfied.

Definition 2.1 *Given a search direction vector* $\mathbf{s}$, *define the symmetric matrix at* $\mathbf{x}_0$

$$B_j(\mathbf{s}, \mathbf{x}_0) = -A^{(j)}(\mathbf{x}_0)^{-\frac{1}{2}}\left(\sum_{i=1}^{n} s_i A_i^{(j)}\right)A^{(j)}(\mathbf{x}_0)^{-\frac{1}{2}} \quad (1 \leq j \leq q). \tag{2.1}$$

Since the standing point $\mathbf{x}_0$ is an interior point of $\mathcal{R}$, $A^{(j)}(\mathbf{x}_0) \succ 0$ for each constraint j, and the square root of $A^{(j)}(\mathbf{x}_0)$ exists. Denote the smallest negative (largest positive) eigenvalue of a symmetric matrix B by $\lambda^-_{\min}(\mathbf{B})$ (respectively, $\lambda^+_{\max}(\mathbf{B})$).

Theorem 2.1 *Assume the ray* $\{\mathbf{x}_0 + \sigma\mathbf{s} \mid \sigma \geq 0\}$ $(\{\mathbf{x}_0 - \sigma\mathbf{s} \mid \sigma \geq 0\})$ *intersects the boundary of* $A^j(\mathbf{x}) \succeq 0$ *at the point* $\mathbf{x}_0 + \sigma_+\mathbf{s}$ *(respectively,* $\mathbf{x}_0 - \sigma_-\mathbf{s}$*). Then*

$$\sigma_+^{(j)} = 1/\lambda^+_{\max}(B_j(\mathbf{s}, \mathbf{x}_0)) \tag{2.2}$$

$$\sigma_-^{(j)} = -1/\lambda^-_{\min}(B_j(\mathbf{s}, \mathbf{x}_0)). \tag{2.3}$$

Proof: We show (2.2). $A^{(j)}(\mathbf{x}) \succ 0$ when $\mathbf{x}$ is in the interior of $\mathcal{R}$, but on the boundary it must have at least one zero eigenvalue. Then

$$\begin{aligned}\sigma_+^{(j)} &= \max\{\sigma \mid \sigma > 0,\ A^{(j)}(\mathbf{x}_0 + \sigma\mathbf{s}) \succeq 0\} \\ &= \min\{\mu \mid \mu > 0, \\ &\qquad \det[A^{(j)}(\mathbf{x}_0 + \mu\mathbf{s})] = 0\}\end{aligned} \tag{2.4}$$

Now, when $\mu > 0$

$$\begin{aligned}
& \det[\mathrm{A}^{(j)}(\mathbf{x_0} + \mu \mathbf{s})] = 0 \\
\iff & \det[\mathrm{A}^{(j)}(\mathbf{x_0}) + \mu \sum_{i=1}^{n} s_i \mathbf{A}_i] = 0 \\
\iff & \det[\frac{1}{\mu}\mathbf{I} + \mathrm{A}^{(j)}(\mathbf{x_0})^{-\frac{1}{2}} \\
& (\sum_{i=1}^{n} s_i \mathbf{A}_i^{(j)}) \mathrm{A}^{(j)}(\mathbf{x_0})^{-\frac{1}{2}}] = 0 \\
\iff & \det[\frac{1}{\mu}\mathbf{I} - \mathrm{B}_j(\mathbf{s}, \mathbf{x_0})] = 0 \\
\iff & \frac{1}{\mu} \textit{ is an eigenvalue of } \mathrm{B}_j(\mathbf{s}, \mathbf{x_0}).
\end{aligned}$$

This and (2.4) give

$$\begin{aligned}
\sigma_+^{(j)} &= \min\{\mu \mid \mu > 0, \frac{1}{\mu} \textit{ is an eigenvalue of } \mathrm{B}_j(\mathbf{s}, \mathbf{x_0})\} \\
&= \min\{1/\lambda \mid \lambda > 0, \lambda \textit{ is an eigenvalue of } \mathrm{B}_j(\mathbf{s}, \mathbf{x_0})\} \\
&= 1/\lambda_{max}^{+}(\mathrm{B}_j(\mathbf{s}, \mathbf{x_0}))
\end{aligned}$$

The proof of (2.3) is similar. □

Corollary 2.1 *An LMI constraint* $\mathrm{A}^{(j)}(\mathbf{x_0}) \succ 0$ *is bounded in the direction* $\mathbf{s}$ *from* $\mathbf{x_0}$ *if and only if the matrix* $\mathrm{B}_j(\mathbf{s}, \mathbf{x_0})$ *has a strictly positive eigenvalue.*

Proof: $\sigma_+^{(j)} > 0$ exists if and only if the feasible region satisfied by $\mathrm{A}^{(j)}(\mathbf{x_0}) \succ 0$ is bounded in the direction $\mathbf{s}$ from $\mathbf{x_0}$. The result follows from Theorem 2.1 □

By (2.2) and (2.3), we find the distances $\sigma_+^{(j)}$ ($\sigma_-^{(j)}$) by computing $\lambda_{min}^{+}(\mathbf{B})$ ($\lambda_{max}^{-}(\mathbf{B})$). Let σ_+ (σ_-) be the finite distances of $\mathbf{x_0}$ to the boundary of the bounded feasible region $\mathcal{R}$ in the direction $\mathbf{s}$ ($-\mathbf{s}$). By (2.2) and (2.3) these are given by

$$\sigma_+ = \min\{\sigma_+^{(j)} \mid 1 \le j \le q\} \quad (2.5)$$

$$\sigma_- = \min\{\sigma_-^{(j)} \mid 1 \le j \le q\} \quad (2.6)$$

SSH Algorithm

Initialization Denote the index set of identified constraints by $\mathcal{J}$ and set $\mathcal{J} = \emptyset$. Choose an interior (standing) point $\mathbf{x_0}$ of $\mathcal{R}$. Calculate $\mathrm{A}^{j}(\mathbf{x_0})^{-1/2}$ for $1 \le j \le q$.

Repeat

Search Direction: Choose $\bar{\mathbf{s}}$ with n entries from N(0,1) and compute $\mathbf{s} = \bar{\mathbf{s}}/\| \bar{\mathbf{s}} \|_2$ to generate a random point $\mathbf{s}$ uniformly on the surface of the unit hypersphere

$$S^{n-1} = \{\mathbf{x} \in \mathbf{R}^n \mid \| \mathbf{x} \|_2 = 1\}.$$

Hitting Step: Calculate $\mathrm{B}_j(\mathbf{s}, \mathbf{x_0})$ and find $\sigma_+^{(j)}$, $\sigma_-^{(j)}$ for $1 \le j \le q$. Calculate $\sigma_+ = \min\{\sigma_+^{(j)} \mid 1 \le j \le q\}$ and $\sigma_- = \min\{\sigma_-^{(j)} \mid 1 \le j \le q\}$. For $1 \le k \le q$, if $\sigma_+^{(k)} = \sigma_+$ or $\sigma_-^{(k)} = \sigma_-$ and $k \notin \mathcal{J}$, set $\mathcal{J} = \mathcal{J} \cup \{k\}$.

Until a stopping rule holds.

Since $\mathbf{x_0}$ is fixed, we only compute $\mathrm{A}^{j}(\mathbf{x_0})^{-\frac{1}{2}}$ once throughout the detection process. After the termination of the SSH algorithm, all LMI's in the set $\mathcal{J}$ are declared necessary.

Theorem 2.2 *In an iteration of the SSH algorithm, for any k, if $\sigma_+^{(k)} = \sigma_+$ or $\sigma_-^{(k)} = \sigma_-$, then $A^{(k)}(\mathbf{x}) \succeq 0$ is necessary with probability 1 in a finite number of iterations.*

Proof: The boundary $\mathcal{S}$ of $\mathcal{R}$ has dimension $n-1$. We assumed in Section 1.0.1 that there are no *duplicate* constraints or constraints that intersect on a subset of $\mathcal{S}$ of Lebesgue measure greater than zero. This implies that on the boundary $\mathcal{S}$, two or more constraints can coincide on at most a $(n-2)$ dimensional subset of $\mathcal{S}$ of Lebesgue measure zero. Suppose $\sigma_+^{(k)} = \sigma_+$. Since, the probability of simultaneously hitting two constraints in one direction is zero, constraint k satisfies $\sigma_+^{(k)} = \sigma_+$ uniquely. Then, $\mathbf{x_0} + \sigma_+\mathbf{s}$ is on the boundary of the $k'th$ constraint and in the interior of each of the remaining constraints. That is, $\lambda_{\min}(A^{(k)}(\mathbf{x_0} + \sigma_+\mathbf{s})) = 0$ and $\lambda_{\min}(A^{(j)}(\mathbf{x_0} + \sigma_+\mathbf{s})) > 0$, for all $j \neq k$.
Choose $\epsilon > 0$ small and define $\mathbf{x}' = \mathbf{x_0} + (\sigma_+ + \epsilon)\mathbf{s}$ so that

$$\begin{aligned} \lambda_{\min}(A^{(k)}(\mathbf{x}')) &< 0 \\ \lambda_{\min}(A^{(j)}(\mathbf{x}')) &> 0, \text{ for all } j \neq k. \end{aligned}$$

Then $\mathbf{x}' \in \mathcal{R}_k$, but $\mathbf{x}' \notin \mathcal{R}$. Hence, $\mathcal{R}_k \neq \mathcal{R}$, which means that constraint k is necessary with probability 1. The proof of the case $\sigma_-^{(k)} = \sigma_-$ is similar. □

It can be shown that SSH detects all necessary LMI constraints with probability 1. Stopping rules for hit-and-run methods ([6],[8],[11]) can be applied to SSH. Finding the best stopping rule for hit-and-run methods is a difficult problem, but there has been recent work in [14].

2.1 Complexity Estimates for SSH

We examine the effect of the sizes m_j of the matrices on SSH to estimate the work required to execute one iteration of SSH. As remarked earlier, we only need to calculate the inverses $A^j(\mathbf{x_0})^{-1/2}$ during the first iteration. Therefore, the main computational effort is in the hitting step, where we compute $\lambda_{\max}^+$ and $\lambda_{\min}^-$ of a matrix. The EISPACK guide [24] states: "with regard to considerations of accuracy and reliability, the methods of computing partial eigensystems falls short of the complete eigensystem methods." In our experiments, we found it more accurate to use the *eig* function of MATLAB to first compute all the eigenvalues of $B_j(\mathbf{s}, \mathbf{x_0})$ and then use the *min* and *max* commands to find $\lambda_{\max}^+$ and $\lambda_{\min}^-$. The *eig* function uses the QR-algorithm, together with accuracy-enhancement features.

To generate the search direction, we compute $\mathbf{s} = \bar{\mathbf{s}} / \| \bar{\mathbf{s}} \|_2$. In the hitting step we compute the $m_j \times m_j$ matrix $B_j(\mathbf{s}, \mathbf{x_0})$ and use the *eig* function to find the eigenvalues for each j. Thus, one SSH iteration requires $O(nqm_q^2 + qm_q^3)$ flops, where m_q is the largest m_j. Let $p_j(\mathbf{x_0})$ be the probability that con-

straint j is detected in an iteration of the SSH algorithm from the standing point $\mathbf{x_0}$. Clearly, $p_j(\mathbf{x_0}) = 0$ if and only if constraint j is redundant. By [12], the expected number of iterations $E(\mathbf{x_0})$ for detecting the necessary constraints is given by

$$E(\mathbf{x_0}) = \sum_{j \text{ is necessary}} \frac{1}{p_j(\mathbf{x_0})} - \sum_{k=2}^{q} (-1)^k \sum_{j_1 < j_2 < \ldots < j_k} \frac{1}{(\sum_{r=1}^{k} p_{j_r}(\mathbf{x_0}))}. \quad (2.7)$$

When there is one detection probability $p_j(\mathbf{x_0})$ which is very small relative to the other probabilities, the dominating term in the expression is $1/p_j(\mathbf{x_0})$, and the expected number of iterations is approximately $1/p_j(\mathbf{x_0})$.

We note that SSH does not solve the redundancy identification problem. It has obvious application to the problem of classification of LMI constraints as redundant or necessary. In the SSH algorithm, constraints with indices in the set $\mathcal{J}$ are necessary while those with indices not in $\mathcal{J}$ are considered redundant. We note that there is a possibility of some error in the classification since a constraint may be incorrectly classified as redundant. In an SDP problem, where constraints represents restriction on resources, it is valuable information to know that a particular constraint is necessary.

3 Application of SSH to Positive Semidefinite Programming

In this section we study the application of SSH to positive semidefinite programming. Redundant LMI constraints are not necessary for the solution process of an SDP. They only add to the computational time. Moreover, it may not be necessary to identify all necessary constraints in solving an SDP. Constraints far from the optimal point may have not been identified as necessary, but the optimal solution may still be feasible with respect to their LMI's.

Given an SDP, we take the *analytic center* [18] defined by the LMI constraints as the standing point for a number of iterations. In this way we obtain a reduced set of the original LMI's. Existing tools (e.g., SDPSOL), can be applied to the reduced set of constraints $\mathcal{J}$ to find a test optimal solution. We describe our *recursive* method:

- If the resulting optimal solution is feasible for all undetected LMI's, then it is an optimal solution of the original SDP.
- If the solution violates a set of undetected LMI's, then the violated LMI's are added to the constraint set $\mathcal{J}$ and the new reduced problem is solved again. At least one of the violated constraints is nonredundant [8].
- The procedure is repeated, until a feasible optimal solution is found.

Examples of this are given in Table 2. For each example, the CPU time (in seconds) taken to solve each reduced problem, and the optimal objective value of the reduced problem are recorded. Let N_{red}= the number of constraints in the reduced SDP, and N_{vio}= the number of constraints violated at the optimal solution of the reduced problem.

Sub-Problem	Löwner 5			
	N_{red}	Time	Obj. Value	N_{vio}
1st	6	1.52	2.3361	5
2nd	11	1.03	2.4778	0
	Löwner 6			
	N_{red}	Time	Obj. Value	N_{vio}
1st	3	2.05	2.6285	5
2nd	8	1.35	2.6532	1
3rd	9	1.30	2.6533	1
4th	10	1.33	2.6537	0

Table 2: SDP solution times and objective values for two reduced problems.

The times taken to solve the original SDP's, i.e., Löwner 5, with 1001 constraints, and Löwner 6 with 5001 constraints are **72.1** and **419.9** seconds respectively. The times (SSH time) taken to run SSH with Löwner 5 and Löwner 6 are **52.0** and **233.2** respectively. Table 2 and the SSH times give the total times for solving Löwner 5 and Löwner 6 with the use of SSH as $\mathbf{54.6} = \mathbf{52.1} + \mathbf{1.5} + \mathbf{1.0}$ and $\mathbf{239.3} = \mathbf{233.2} + \mathbf{2.1} + \mathbf{1.4} + \mathbf{1.3} + \mathbf{1.3}$ seconds respectively, which are less than the times given above for solving the original problems. Most of the effort is expended in reducing the size of the problem.

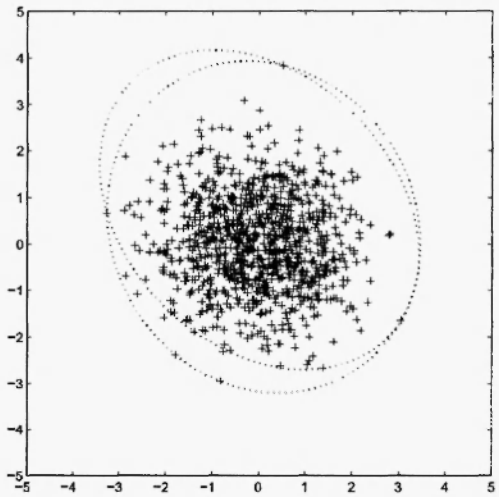

Figure 1: The Löwner ellipsoids of Löwner5 obtained in Table 2.

The 5 violated constraints are clearly identified as the 5 points in the South-West part of Figure 1 that are not contained in the first ellipse. These points are contained within the boundary of the second ellipsoid.

The full implementation of SSH to solving SDP is still under investigation. We believe that a more efficient implementation of SSH is possible, to reduce the SSH time. For example, the analytic center generally isn't an *optimal standing point* from which one expects the smallest number of iterations of SSH.

4 Variations of SSH

In this section, we introduce the *Undetected First Rule* for SSH. This rule is especially useful when there is a high percentage of necessary constraints. We have found it easy to generate such examples by generating random problems in spaces of high dimension. We next introduce the corresponding semidefinite extension of two other well known hit-and-run methods for linearly constrained regions, HD and CD. The corresponding semidefinite extension of HD and CD are called SHD and SCD respectfully.

4.1 SSH with Undetected-First Rule

The direction vector s of an SSH iteration is called *successful* if a new constraint is detected in this direction; otherwise $\mathbf{s}$ is called *unsuccessful*. If a hit point of some necessary constraint is nearer than the closest hit point of the undetected constraints, then $\mathbf{s}$ is an unsuccessful direction. We don't need to calculate the hit points of the other detected constraints. This is the basic idea of the *Undetected-First Rule (UFR)* ([17], [13]). UFR was first suggested in [7] to improve the HD algorithm.

According to the rule, we first calculate the hit points of all the undetected constraints. We then calculate the hit points of the detected constraints one by one. At any time, if the hit point of some detected constraint is smaller than the smallest hit point of the undetected constraints, then $\mathbf{s}$ is unsuccessful; there is no need to calculate the hit points of the remaining detected constraints.

We apply the UFR technique simultaneously in the directions $\mathbf{s}$ and $-\mathbf{s}$. When both directions are *successful*, it is not always necessary to calculate all the hit points of the constraints, and time savings are possible. The SSH method with UFR to find the necessary constraints of the set $\{\mathrm{A}^{(j)}(\mathbf{x}) \succeq 0\}_{j=1}^{q}$ is denoted by SSH(UFR) . In the SSH(UFR) algorithm, $\sigma_+^{(j)}$ ($\sigma_-^{(j)}$) is evaluated only when $\mathrm{B_j}(\mathbf{s}, \mathbf{x_0})$ has a positive (negative) eigenvalue. Let the index set of detected constraints be denoted by $\mathcal{J}$.

SSH(UFR) Algorithm

Replace the hitting step of SSH by:

Hitting Step: Calculate $\mathrm{B_j}(\mathbf{s}, \mathbf{x_0})$ and find $\sigma_+^{(j)}$, $\sigma_-^{(j)}$ for all $j \notin \mathcal{J}$.
Calculate $u_+ = \min\{\sigma_+^{(j)},\ j \notin \mathcal{J}\}$
and $u_- = \min\{\sigma_-^{(j)},\ j \notin \mathcal{J}\}$.
Let $\mathcal{J} = \{j_1, j_2, \ldots, j_N\}$. Set $r = 1$,
POS=FALSE, NEG=FALSE
REPEAT

Calculate $\mathrm{B_{j_r}}(\mathbf{s}, \mathbf{x_0})$ and find $\sigma_+^{(j_r)}$, $\sigma_-^{(j_r)}$
If $\sigma_+^{(j_r)} < u_+$, then POS=TRUE

If $\sigma_-^{(j_r)} < u_-$, then NEG=TRUE

Set r=r+1

UNTIL (((POS=TRUE) and (NEG=TRUE)) or $(r = N+1)$)

IF POS=TRUE and NEG=TRUE, then exit hitting step

ELSE

Calculate

$\sigma_+ = \min\{u_1, \sigma_+^{(j_r)} \mid 1 \le r \le N\}$

$\sigma_- = \min\{u_2, \sigma_-^{(j_r)} \mid 1 \le r \le N\}$.

For $1 \le k \le q$, if $\sigma_+^{(k)} = \sigma_+$ or $\sigma_-^{(k)} = \sigma_-$ and $k \notin \mathcal{J}$, set $\mathcal{J}=\mathcal{J} \cup \{k\}$

Exit hitting step.

Problem	SSH		SSH(UFR)	
	Iter	**Time**	Iter	**Time**
Löwner 1	1023	29.9	1023	17.3
Löwner 2	3260	2600.1	3260	2924.4
Löwner 3	26623	1719.8	26623	1091.5
Löwner 4	18946	1180.4	18946	591.0
Löwner 5	735	1824.3	735	2154.4
Löwner 6	206	4772.3	206	6547.5
Löwner 7	62458	6239.3	62458	1715.1
Löwner 8	31667	2746.1	31667	768.9
Löwner 9	2375	4336.3	2375	5320.5
Löwner 10	72375	9617.2	72375	1945.9

Table 3: Number of iterations and time (in seconds) to find *all* of the necessary constraints

Table 3 compares SSH with SSH(UFR) where both are implemented from the analytic center. The number of iterations and the time (in seconds) for complete detection of the estimated necessary constraints are recorded.

The results of Table 3 for Löwner1, Löwner3, Löwner4, Löwner7, Löwner8 and Löwner10 show that, for certain problems, SSH(UFR) takes less time than SSH for detection. But, in some cases, the cost of implementing UFR in SSH outweighs its benefit. The advantage of SSH(UFR) over SSH increases as both the number of necessary constraints and the number of iterations increase. In an SSH(UFR) iteration, where N constraints have already been detected in previous iterations, we can avoid computing up to $N-1$ hit points of the constraints detected previously. UFR works well when the ratio of the number of necessary constraints to the number of all constraints is near 1.

To reduce computations even further we can reorder the detected constraints in descending order according to their frequency of being detected during the previous iterations, so that we look at the more likely ones first.

Semidefinite Hypersphere Direction (SHD) Method:

SHD is similar to SSH, but in the former, the standing point is not fixed. In an SSH iteration, the two hit points determine a line segment through the standing point. In the next iteration of SHD, a new standing point is chosen uniformly along the line segment of the current iteration. The idea is to move the

standing point randomly about the region $\mathcal{R}$ in order to improve the odds of locating all the constraints. The SHD algorithm is similar to that of SSH, but with the hitting step replaced by:

> **Hitting Step**: Calculate $B_j(s, x_0)$ and find $\sigma_+^{(j)}$, $\sigma_-^{(j)}$ for $1 \le j \le q$.
> Calculate $\sigma_+ = \min\{\sigma_+^{(j)} \mid 1 \le j \le q\}$ and $\sigma_- = \min\{\sigma_-^{(j)} \mid 1 \le j \le q\}$.
> For $1 \le k \le q$, if $\sigma_+^{(k)} = \sigma_+$ or $\sigma_-^{(k)} = \sigma_-$ and $k \notin \mathcal{J}$, set $\mathcal{J} = \mathcal{J} \cup \{k\}$.
> Choose u from U(0,1) and set $x_0 = x_0 + (u(\sigma_+ + \sigma_-) - \sigma_-)s$.

Since x_0 changes, $A^j(x_0)^{-\frac{1}{2}}$ must be computed for each iteration. Therefore, a SHD iteration is $O(nqm_q^2 + qm_q^3)$ flops and computationally more expensive than one of SSH. By [5], the standing points x_0 generated by SHD are asymptotically uniformly distributed over the interior of $\mathcal{R}$.

Semidefinite Coordinate Direction (SCD) Method:

SCD differs from SHD only in the choice of the direction vector s. In SCD, s is chosen randomly from the $2n$ directions e_i parallel to the coordinate axes. If $s = e_i$, where e_i is the canonical unit vector, then

$$B_j(s, x_0) = -A^j(x_0)^{-\frac{1}{2}}\left(\sum_{i=1}^{n} s_i A_i^{(j)}\right)A^j(x_0)^{-\frac{1}{2}}$$

$$= -A^j(x_0)^{-\frac{1}{2}} A_i^{(j)} A^j(x_0)^{-\frac{1}{2}}.$$

The work per iteration of SCD is independent of s and requires $O(qm_q^3)$ flops.

Problem	SSH(UFR) **Time**	SCD **Time**	SHD **Time**
Löwner 1	17.3	23.7	17.9
Löwner 2	2924.4	1165.3	1530.5
Löwner 3	1091.5	31.6	89.9
Löwner 4	591.0	398.7	958.7
Löwner 5	2154.4	4358.6	3334.1
Löwner 6	6547.5	3887.0	5268.5
Löwner 7	1715.1	54.5	349.6
Löwner 8	768.9	2980.1	6580.5
Löwner 9	5320.5	925.0	14597.0
Löwner 10	1945.9	11810.0	16868.0
	Iter	**Iter**	**Iter**
Löwner 1	1023	205	154
Löwner 2	3260	296	375
Löwner 3	26623	83	234
Löwner 4	18946	759	1706
Löwner 5	735	504	385
Löwner 6	206	38	44
Löwner 7	62458	70	464
Löwner 8	31667	3883	8509
Löwner 9	2375	127	1997
Löwner 10	72375	9439	13444

Table 4: Comparison of the time (in seconds) and the number of iterations to find *all* necessary constraints.

The cost per iteration of SSH and SHD increases with increasing n, but not in the case of SCD. SSH has an advantage in that it works with UFR. It is difficult to implement UFR in SHD or in SCD because in these methods all the boundary hit points $x_0 + \sigma_+^{(j)}s$ and $x_0 -$

$\sigma_{\pm}^{(i)}$'s must be calculated in order to determine the next standing point. Table 4 compares SSH hitting from the analytic center with SHD and SCD. It gives the number of iterations and the time (in seconds) for complete detection of the estimated necessary constraints.

Table 4 suggests that SCD performs, on average, better than SSH and SHD on Loewner type problems. SHD was never the best in each of the problems used here, although it has proven to be useful as a tool for finding all the necessary constraints for non-Loewner type problems [18]. SSH(UFR) outperformed SCD and SHD in Löwner1, Löwner5, Löwner8 and Löwner10. The performance of SSH(UFR) is dependent on the choice of the standing point.

5 Conclusion

We have introduced three Monte Carlo methods for detecting redundant LMI constraints: SSH, SHD and SCD. The cost per iteration for SSH is $\mathbf{O}(nqm_q^2 + qm_q^3)$, and this is clearly polynomial per iteration.

We have shown experimentally that we can obtain superior execution times for solving SDP problems using these techniques. We have demonstrated that these are useful methods for the classification of LMI constraints as redundant or necessary. Thus the Monte Carlo methods can be used to solve SDP's with large numbers of LMI constraints. The use of the Undetected-First Rule (UFR) reduced the number of computations in the hitting step in some instances. The savings rise as the number of necessary constraints increases. We have demonstrated that this gives a practical method for solving Löwner ellipsoid problems. Problems requiring further research are the development of new stopping rules for SSH and the choice of a standing point.

Acknowledgements

This research was partially supported by an NSERC GR-5 and NSERC equipment grants. The authors thank Richard J. Caron for his helpful suggestions in the writing of this paper.

References

[1] F. Alizadeh, "Interior Point Methods in Semidefinite Programming with Applications to Combinatorial Optimization", *SIAM J. Optim.*, Vol. 5, no. 1, 1995, pp. 13-51.

[2] F. Alizadeh "*SDPpack - Version 0.9 Beta for Matlab 5.0*", User's Guide, June 1997.

[3] E. Andersen and K. Andersen, "Presolving in linear programming", *Math. Programming*, Vol. 71, 1995, pp. 221-245.

[4] S. J. Benson, Yinyu Ye and Xiong Zhang, "Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization", *SIAM Journal on Optimization*, Vol. 10, no. 2, 2000, pp. 443-461.

[5] C. Belisle, A. Boneh and R. Caron, "Convergence properties of hit-and-run samplers", *Stochastic Models*, Vol. 14, no. 4, 1998.

on Matrix Analysis and Applications, Vol. 19, 1998, pp. 499-533.

[29] H. Wolkowicz, "Some applications of optimization in matrix theory", *Linear Algebra and its Applications*, Vol. 40, 1981, pp. 101-118.

[30] S. -P. Wu and S. Boyd, "*SDPSOL - A Parser/Solver for Semidefinite Programming and Determinant Maximization Problems with Matrix Structure*", User's Guide, Version Beta, May 1996.